A Primer on NoSQL Databases for Enterprise Architects: The CAP Theorem and Transparent Data Access with MongoDB and Cassandra

Fumbeya Marungo
Johns Hopkins University
fmarung1@jhu.edu

***Abstract*—MongoDB and Apache Cassandra are the dominant "Not Only SQL" (NoSQL) database management systems for persisting structured records. Moreover, the pair are respectively in the top-five and top-ten of database management systems generally. Therefore this work seeks to present the two leading systems, along with the underlying principle of the CAP Theorem, in the context of creating transparent data access tiers capable of supporting flexible enterprise architectures.**

1. Introduction

The goal of this work is to present the tightly related principles of the CAP Theorem, BASE semantics, and ACID transactions, along with the two leading "Not Only SQL" (NoSQL) database management systems MongoDB [22] and Apache Cassandra [1] in the context of enterprise architecture.

The interplay of CAP, BASE, and ACID are not only conceptual underpinnings of NoSQL systems, but also have fundamental implications on location, transactional, and redundancy decisions across the enterprise architecture.

This work focuses on MongoDB and Cassandra as the dominant NoSQL leaders; respectively receiving 50% and 15% of NoSQL LinkedIn Skills mentions [5]. More broadly, MongoDB is ranked in the top-five, and Cassandra in the top-ten in popularity across all database systems [26].[1] Each system offers an alternative data model to that of the traditional relational database management system (RDBMS), along with distinct straight-forward replication and transaction approaches.[2]

Given the level of popularity of MongoDB and Cassandra there is a need for examination of their potential applications to enterprise architecture. For example, the current version of The Open Group Architecture Framework (TOGAF) [27] is from 2011; it does not mention either NoSQL systems, the database models they provide, or the CAP Theorem. This work presents the features of MongoDB and Cassandra that are relevant to the development of data access tiers.

In the Software Engineering View of TOGAF [27], the data access tier of a five-tier software architecture provides an intermediary interface between the data store tier and one or more application tiers across the enterprise. The tier it serves persisted data classes to the application architecture while hiding the complexity of the data architecture.

To define the requirements for hiding complexity, the TOGAF Software Engineering View outlines a set of *transparency* properties drawn from the ISO Reference Model for Open Distributed Processing (RM-ODP). The purpose of a transparency is to specify a class of problems a component safely hides from its clients [21].

Thus, application tier clients can use the data access tier to access persisted objects while safely meeting service level agreement (SLA) requirements. Common requirements include providing: a) 99.9% availability with database commits for human user systems within 5 ms [17], and b) replicated redundancy sufficient to recover from site-wide or regional failure for business continuity planning.

This work examines relevant features of MongoDB and Cassandra in enabling the data access tier to address six RM-ODP transparencies[3] .

2. The CAP Theorem, ACID and BASE

The technical motivation for developing NoSQL systems grew from the CAP Theorem — also called Brewer's Conjecture [11]. The three factors in the theorem are as follows:

- *Consistency* is the guarantee that simultaneous reads from separate locations return the same value. That is the system does not return stale or conflicting data.
- *Availability* is both the responsiveness of a system in normal operations and the likelihood of outages.
- *Partition* is the number, fallibility, and performance of independent components in the system, and the

1. A third system, Redis [4], is roughly tied with Cassandra. However, it stores scaler key-value pairs rather than structured data. Combined the three systems represent 80% of LinkedIn mentions [5].

2. For examples this work uses the sample database provided by the Eclipse Foundation's Business Intelligence and Reporting Tool (BIRT) Project [2]. IBM is a BIRT project sponsor and the project underlies the company's Tivoli reporting software.

3. There are six RM-ODP transparencies, two are not applicable to this work: a) *location transparency* is typically provided by a separate naming or directory service, and b) *persistence transparency* concerns issues of stateful object lifecycle management, which is optional and distinct from the data access tier's stateless operations on the data store.

URI: http://hdl.handle.net/10125/50472
ISBN: 978-0-9981331-1-9

HICSS

network that connects them. Modern enterprise architectures are distributed — i.e. partitioned — by default.

By the modern construct of the CAP Theorem: in the presence of partitioning, choose between consistency or availability [7].

Choosing consistency over availability prioritizes receiving a guarantee that a read does not return stale or conflicting data while accepting slower overall responsiveness due to the added coordination burden for each query, and the risk of an indefinite outage due to a network or system component failure.

Choosing availability over consistency gives precedence to receiving an immediate read even if the results are stale or inaccurate, and to accepting an acknowledgment of a write even if the change may not be visible throughout the system, and may be subsequently lost.

2.1. ACID

In traditional database theory, ACID transactional properties [14], [15] are the dominant paradigm. The terms stand for:

- *Atomic*, all of a transaction's writes are committed or rolled back as one unit. There are no partial writes.
- *Consistent*, a transaction does not see interim changes of another transaction. That is, the transaction only reads committed data.
- *Isolation*, once a transaction reads or writes data it is guaranteed that the data is not changed by a concurrent transaction until the commit or rollback.
- *Durable*, a successful transaction is recoverable if the system fails.

2.1.1. ACID and The CAP Theorem. CAP consistency and ACID consistency are different. CAP consistency concerns simultaneous queries returning the same value regardless of location. ACID consistency concerns hiding a transaction's writes from concurrent transactions until a commit.

For example, in a bank transfer, reading the first part of the transfer before the second is performed violates ACID consistency, but CAP consistency is not violated so long as the value read is the same. Conversely, reading the correct state of both accounts before the transfer at one location, and simultaneously reading the correct state after the transfer is ACID consistent, but reading two different values at the same time from separate locations is not CAP consistent. For the remainder of this paper, consistency refers to the CAP concept.

2.1.2. Distributed ACID Sacrifices Availability. Gilbert and Lynch [13] provides a formal proof of the CAP Theorem. The work further proves that providing ACID transactions requires consistency, and thus sacrifices availability in a distributed system. The more partitioning in the system, the greater the loss of availability due to distributed ACID guarantees.

The loss of availability due to distributed ACID transactions is not a theoretical result. Each operation of the system is slowed by transaction management over the network. Further, the system is vulnerable overall to a network or component failure.

The true cost of preserving ACID guarantees is often hidden. Slowed responses are simply accepted. Manual interventions mask the outages. Moreover, during intervention, committed writes are sometimes lost. Therefore, in reality durability is not an absolute guarantee.

2.2. Abandoning Distributed Transactions

The "apostate's" argument in Helland [16] asserts that the cost in availability makes ACID-compliant distributed transactions infeasible in designing scalable systems.

In Helland [16], scalable information system architectures consist of entities, where each entity is identifiable by a key and is stored as a single unit at an individual machine. Each machine in the system can only perform ACID-compliant operations on its local entities. The entities then exchange messages asynchronously — i.e. outside of transactional guarantees.

Helland [16] applies beyond an individual database management system; it impacts process specification across architectural phases. Process models should not rely on transactional enforcement beyond the scope of a single entity. Thus, verification activities should be incorporated into process specifications involving multiple persisted entities in lieu of relying upon transactional guarantees.

2.3. Replication and ACID Relaxation

Replication in Helland [16] cannot occur within a full transaction; by definition replicating writes is a distributed activity.

Thus a key implication of the CAP Theorem on enterprise architectures is that the combination of ACID transactions and redundancy in fact produces system fragility.

To fulfill replication requirements, a system must relax ACID guarantees in order to enable replication without losing availability, as well as adhere to Helland [16] by limiting transactional scope to a single record.

2.4. BASE

Neither of the CAP Theorem's alternative are permissible in reality. A system cannot be prone to outages, or regularly provide stale and inaccurate data. BASE semantics [12] are a representation of the real-world tradeoffs necessary arising from ACID relaxation choices.

The meaning of the acronym is as follows [24]:

- *Basic Availability* is the expected responsiveness of the system. It comprises normal and peak demand response rates, as well as the likelihood of an outage.
- *Soft-state* is the data that may be lost. This represents the uncertainty of reads or writes due to a failure.
- *Eventually consistency* is stale or conflicting data that has not been updated or reconciled. This represents the uncertainty during normal operations due to the separation and partitioning of components.

Basic availability is beneficial; soft-state and eventual consistency are detrimental. Thus, the goal of ACID relaxation is to increase the former, or decrease the latter two.

3. The Emergence of NoSQL Databases

NoSQL database management systems arose in response to limitations of RDBMS due the needs to perform join operations and maintain referential integrity: [20]

- *Object-Relational Impedance Mismatch* – The difficulty in matching RDBMS entity-relational models to software classes is well-known. A customer order in the logical data model of Figure 1 is not one cohesive unit. To create a customer order, a row in the `Orders` table must be inserted before the related rows in the `OrderDetails` table; the reverse must be done to delete the order. A wider concern is that modifying a `Customer` row has the hidden side effect of overwriting all previous orders.
- *Distributed Joins* – As a RDBMS grows, rows become more distributed as tables are placed in separate locations, and are further subdivided through sharding. Performing joins is a costly operation locally. Without careful administration, joins become far costlier as distributed transactions that are subject to the CAP Theorem. Moreover, full ACID compliance is necessary to preserve referential integrity. For example, the query returning a customer order in Listing 1 must enforce integrity over six rows; at scale, it is difficult to ensure the rows are co-located in a complex database.
- *Complex Replication* – RDBMS replication is complex because, not only must individual rows be replicated, but also referential integrity must be preserved in the process. As a result, there is no standard approach to replication. MySQL, for example, has two separate and distinct clustering engines, with different transactional guarantees. Thus the true ACID behavior of a RDBMS deployment with replication is often specific to the particular RDBMS and a large number of custom settings.

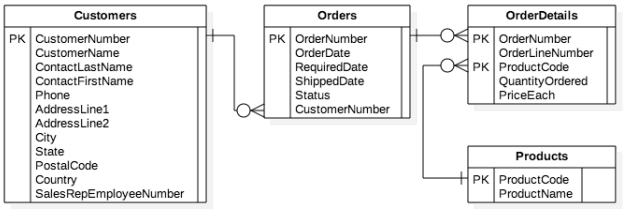

Figure 1: Tables related to a customer order from the BIRT sample database [2].

3.1. NoSQL Characteristics

Characteristics of MongoDB, Cassandra, and other NoSQL systems for avoiding the limitations of joins and referential integrity include:

- *Nested Hierarchal Record Structures*[4] – Self-contained records store related data as a unit without the need for joins.
- *Single-Record Transactions* – The systems limit transactional guarantees to operations over one record.
- *Simplified Replication* – Each NoSQL system offers one record replication method with a small set of tuning configurations.

Thus, NoSQL systems inherently implement Helland [16]. Each server manages transactional operations on its local copies of records. Each record is individually replicated without the need to maintain referential integrity. NoSQL systems are classified by their approach to replication: a) Consistency Partition (*CP*) systems — such as MongoDB — prioritize returning the same value across locations, and b) Availability Partition system (*AP*) — such as Cassandra — maintain responsiveness by returning results from low latency replicas.

To enable reliable replication, each NoSQL system also explicitly relaxes ACID protections, while providing BASE tuning settings to manage the degree of relaxation, for example: a) MongoDB relaxes durability to permit rollbacks of inconsistent writes, and b) Cassandra relaxes isolation to remove the need to avoid collisions by disparate servers.

4. MongoDB

MongoDB is an open-source system by the eponymous company; it is often deployed as the data storage component of the MEAN JavaScript-based three-tier web application stack. The remainder of the stack consists of: Express.js for application logic, Angular.js for the user interface within the browser client, and Node.js to execute Express.js components on the server. Thus the system is designed as a straight-forward means to persist user interactions.

4. Except in the case of scalar key-value stores.

```
SELECT o.OrderNumber, OrderDate, RequiredDate, ShippedDate, Status,
  c.CustomerNumber, CustomerName, ContactLastName, ContactFirstName,
  Phone, AddressLine1, AddressLine2, City, State, PostalCode,
  Country, SalesRepEmployeeNumber, OrderLineNumber, QuantityOrdered,
  p.ProductCode, ProductName, PriceEach
FROM Orders o JOIN OrderDetails od ON o.OrderNumber = od.OrderNumber
  JOIN Customers c ON o.CustomerNumber = c.CustomerNumber
  JOIN Products p ON od.ProductCode = p.ProductCode
WHERE o.OrderNumber = 10102
ORDER BY OrderLineNumber

ORDERNUMBER |... |CUSTOMERNUMBER... |ORDERLINENUMBER |PRODUCTCODE...
------------|----|------------------|----------------|-----------...
10102       |... |181           ... |1               |S18_1367   ...
10102       |... |181           ... |2               |S18_1342   ...
```

Listing 1: SQL statement to retrieve the BIRT customer order.

```
> use birt
switched to db birt
> db.orders.insertOne({
...    "_id": 10102, "date": ISODate("2011-01-10"),
...    "required_date": ISODate("2011-01-18"),
...    "shipping_date": ISODate("2011-01-14"),
...    "status" : "shipped",
...    "customer": {
...      "customer_number": 181, "customer_name": "Vitachrome Inc.",
...      "contact_last_name": "Frick",
...      "contact_first_name": "Michael",
...      "phone_no": "415-555-1450",
...      "address_line_1": "2678 Kingston Rd.",
...      "address_line_2": "Suite 101",
...      "city": "NYC", "state": "NY", "postal_code": "10022",
...      "country": "USA", "sales_rep_employee_number" : 1286
...    },
...    "order_details": [{
...        "product": {
...          "product_code": "S18_1367",
...          "product_name":
...            "1936 Mercedes-Benz 500K Special Roadster",
...        },
...        "quantity": 41, "unit_price": 43.13
...      },{
...        "product": {
...          "product_code": "S18_1342",
...          "product_name": "1937 Lincoln Berline",
...        },
...        "quantity": 39, "unit_price": 95.55
...      }
...    ],
... })
{ "acknowledged" : true, "insertedId" : 10102 }
> db.orders.findOne({"_id": 10102})
```

Listing 2: Adding the customer order to MongoDB using the JavaScript shell. The `birt` database and `orders` collection are created automatically.

4.1. A JSON Database

A *database* in MongoDB contains *collections*. The records in each collection are in a variant of JavaScript Object Notation (*JSON*) called Binary JSON (*BSON*). BSON has an extended typing system, but it is essentially JSON. We refer to the records as JSON documents in the remainder of this paper.

JSON is a language-independent ECMA-standard [3] data format for object interoperable representation. The protocol that is popular in Internet development due to its simplicity and compatibility with JavaScript.

Listing 2 adds the BIRT customer order to an `orders` collection, and then retrieves it from the database.

The listing highlights that MongoDB is immediately ready to persist objects. If the `birt` database and `orders` collection do not exist, the system creates them. The database system is entirely schema-less. Each document can have an arbitrary structure. Also, the JSON document is in a simple, human readable format. In addition, the queries and responses are JSON documents as well. Querying the database is declarative, and does not involve complex manual programming.

Listing 2 uses the `mongo` JavaScript interactive shell included with the database. In addition there is official support for Java, Python, C#, C/C++, PHP, Ruby, and other programming languages. Enterprise architects who have worked with XML, and at least one scripting language, in the past should find the system easy to use.

4.2. Consistent Replication

MongoDB manages replication through a group of nodes called a *replica set*. In normal cases, a replica set represents an entire collection. Each member of the replica set has a full copy of the entire collection along with any search indexes declared on the collection. With very large collections, or to distribute the work load, the collection is divided evenly, or *sharded*; each replica set handles a shard rather than an entire collection.

If the collection is not sharded, ACID operations can occur over multiple records within a collection because each replica set member has a copy of the entire collection. Once a collection is divided into shards, atomic operations can only occur on a single document, because there is no longer a guarantee that different documents within the collection share the same replica set. Therefore, MongoDB follows the approach in Helland [16], transactional guarantees are limited to operations of a node on its local replicas.

When the replica set begins, the voting members elect a *primary* member by majority vote. The primary handles writes for the collection or shard. With each write, the primary writes an update entry to a local *oplog* of the changes to a record(s) and related indexes. Each *secondary* replicates writes by copying over the entries.

In this approach the primary is the "true" state of the data. While the secondaries can lag behind and become consistent eventually. Under the default settings all reads are also routed to the primary.

Therefore, MongoDB is a CP system because, both reads and writes are performed at a single source.

4.3. BASE Considerations: Regional Availability, and Soft-State Durability Relaxation

Clients far from the primary node (in network distance) have considerably lower basic availability due to added network latency. Each write must be handled by the primary node. However, it is possible to increase responsiveness for reads by configuring a query's *read preference* to permit routing to nearby secondaries.

By BASE semantics, changing the read preference introduces eventual consistency. To avoid excessive stal-

eness, MongoDB also provides query settings to limit the lag time of the nodes permitted to perform the read.

4.3.1. Primary Failure Soft-State Non-Durability. By default the primary acknowledges a write immediately after making it locally durable. This increases basic availability by reducing the wait time for a response, however, the acknowledgment does not confirm replication.

If the primary fails, or is no longer part of a majority of the replica set due to a connectivity loss, the remaining majority elect a new primary. Any write that was not replicated to at least one of the participants in the election is rolled back.

Therefore, MongoDB relaxes ACID durability during failover; acknowledged writes may be lost after a primary becomes inaccessible to a replica set majority.

To monitor the status of the replica set, members exchange heartbeat messages. Within 10-30 seconds, a majority of the replica set can detect that the primary is no longer accessible; concurrently, the primary (if it is operating) discovers it can not access a majority and steps down.

When the former primary rejoins the replica set, it rolls back the writes it acknowledged which the electing majority did not capture.

4.3.2. Increasing Durability. To ensure a write is durable, the query's *write concern* setting must direct the primary to acknowledge a write only after durable replication by a majority of the replica set.

To prevent reading data that may be rolled back a query's *read concern* setting must direct the primary to linearize all reads. That is the primary only reads data that are confirmed to be durably replicated by a majority before the read was submitted. None of the results are subject to a future rollback.

By BASE principles, the reduction in soft-state uncertainty comes at the cost of basic availability. By receiving acknowledgments for writes only after replication to a majority, and linearizing reads, queries have fully ACID-compliant replication. However, these setting do not circumvent the CAP Theorem. Writes have latencies arising from the primary waiting for acknowledgment from the last member necessary to form a majority. Reads require a time out setting to avoid the possibility of an indefinite wait time.

5. Cassandra

Cassandra began as a Facebook project to support inbox search [18]. The Apache open-source project has evolved to provide an environment similar to a RDBMS without the `JOIN` operator. Since 2015, the system's standard programming interface is a custom SQL-like language, Cassandra Query Language (*CQL*). Like an RDBMS, drivers are available for many languages, as well as for Open Database Connectivity (ODBC).

Orders		
PK	OrderNumber	PARTITION KEY
	OrderDate	STATIC
	RequiredDate	STATIC
	ShippedDate	STATIC
	Status	STATIC
	CustomerInfo	STATIC MAP
PK	OrderLineNumber	CLUSTERING COLUMN
	ProductInfo	MAP
	QuantityOrdered	
	PriceEach	

Figure 2: A Cassandra table for the BIRT sample database's business order [2].

5.1. A Table Partition Record with CQL

A database in Cassandra is called a *keyspace*. The keyspace contains a set of *tables*. Each record in Cassandra is a *partition* of its table with rows grouped by a column(s) called the *partition key*. The rows within the partition are identified by the *clustering* column(s) that are unique within the partition.

The *primary key* is the union of the partition key and clustering columns. This is a confusing point because, the primary key uniquely identifies the row, but the record is not one row, it is a partition.

Static columns store partition-wide data that is not part of the partition key. The remainder of the columns store data for each row.

Cassandra permits *collection* typed columns which are maps, sets, or lists. Map or set columns perform well; however, the documentation advises against lists because of performance concerns.

5.1.1. The CQL `Orders` Table. A partition in CQL is akin to a `LEFT OUTER JOIN` on a single row from the left table in SQL. The partition key and static columns correspond to the one row in the left table, and the clustering columns and non-static data columns are from the matching zero to many rows in the right table.

Figure 2 is an `Orders` table in Cassandra. The partition key, `OrderNumber`, identifies the order, with the related parent data in static columns. The clustering column, `LineItemNumber`, identifies the line item within the order, with the remaining columns containing the child data for each line item.

Respective static and non-static columns in the `Orders` table contain the customer and product information originally in separate tables of the BIRT sample relational database.

Therefore, each partition in the Cassandra `Orders` table conforms to Helland [16]. One record holds the entire customer order record.

A review of the CQL in Listing 3 provides a view into the language and Cassandra's record structure. Section 5.2 covers the `WITH REPLICATION` declaration.

Line 9 and Line 10 declare the respective `CustomerInfo` and `ProductInfo` map columns. Two limiting factors of collection columns are: the need to declare the type of a collection column's contents,

```
1  -- create a birt keyspace
2  CREATE KEYSPACE birt WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_factor' : 1};
3    --- WITH REPLICATION = {'class' : 'NetworkTopologyStrategy', 'dc1' : 3, 'dc2' : 3, 'dc3' : 3};
4  USE birt;
5
6  -- create the orders table
7  CREATE TABLE Orders (
8    OrderNumber int, OrderDate timestamp STATIC, RequiredDate timestamp STATIC, ShippedDate timestamp STATIC, Status varchar STATIC,
9    CustomerInfo MAP<varchar, varchar> STATIC,
10   OrderLineNumber int, ProductInfo MAP<varchar, varchar>, QuantityOrdered int, PriceEach decimal,
11   PRIMARY KEY (OrderNumber,OrderLineNumber));
12
13 -- an 'upsert' that inserts only top-level partition data, because the clustering column is not included
14 INSERT INTO Orders
15   (OrderNumber, OrderDate, RequiredDate, ShippedDate, Status, CustomerInfo)
16 VALUES
17   (10102, '2011-01-10', '2011-01-18', '2011-01-14', 'In process',
18   {
19       'customer_number': '181', 'customer_name': 'Vitachrome Inc.',
20       'contact_last_name': 'Frick', 'contact_first_name': 'Michael',
21       'phone_no': '415-555-1450',
22       'address_line_1': '2678 Kingston Rd.',
23       'address_line_2': 'Suite 101',
24       'city': 'NYC', 'state': 'NY', 'postal_code': '10022', 'country': 'USA',
25       'sales_rep_employee_number' : '1286'
26   });
27
28 SELECT * FROM Orders WHERE OrderNumber = 10102
29  ordernumber | orderlinenumber | orderdate  |...| status     | customerinfo                                  | priceeach | productinfo...
30 -------------+-----------------+------------+...+------------+-----------------------------------------------+-----------+------------...
31        10102 |            null | 2011-01-10 |...| In process | {'address_line_1': '2678 Kingston Rd.'...  |      null |        null...
32
33 -- upsert that inserts the first row in the partition, note the previous `null' row disappeared
34 INSERT INTO Orders
35   (OrderNumber, OrderLineNumber, ProductInfo, QuantityOrdered, PriceEach)
36 VALUES
37   (10102, 1,
38   {
39     'product_code': 'S18_1367',
40     'product_name': '1936 Mercedes-Benz 500K Special Roadster'
41   },
42   41, 43.13);
43
44 SELECT * FROM Orders WHERE OrderNumber = 10102
45  ordernumber | orderlinenumber | orderdate  |...| status     | customerinfo                                  | priceeach | productinfo                ...
46 -------------+-----------------+------------+...+------------+-----------------------------------------------+-----------+----------------------------...
47        10102 |               1 | 2011-01-10 |...| In process | {'address_line_1': '2678 Kingston Rd.'... |     43.13 | {'product_code': 'S18_1367'...
48
49 -- upsert that both inserts the second row , and update the Status (a static column),
50 INSERT INTO Orders
51   (OrderNumber, OrderLineNumber, Status, ProductInfo, QuantityOrdered, PriceEach)
52 VALUES
53   (10102, 2, 'Shipped',
54   {
55     'product_code': 'S18_1342',
56     'product_name': '1937 Lincoln Berline'
57   },
58   39, 95.55);
59
60 SELECT * FROM Orders WHERE OrderNumber = 10102
61  ordernumber | orderlinenumber | orderdate  |...| status  | customerinfo                                  | priceeach | productinfo                ...
62 -------------+-----------------+------------+...+---------+-----------------------------------------------+-----------+----------------------------...
63        10102 |               1 | 2011-01-10 |...| Shipped | {'address_line_1': '2678 Kingston Rd.'... |     43.13 | {'product_code': 'S18_1367'...
64        10102 |               2 | 2011-01-10 |...| Shipped | {'address_line_1': '2678 Kingston Rd.'... |     95.55 | {'product_code': 'S18_1342'...
```

Listing 3: CQL statements to create, populate, and select from a table for orders.

and the inability to nest collections — i.e. a collection cannot not contain another collection[5]. As a result of the former, the customer number in the customer information is a `varchar`.

In the primary key declaration (Line 11), the first column, is the partition key, and the remainder are the clustering columns. To declare Multi-column *compound* partition keys, enclose the opening columns in parenthesis.

5.1.2. Upserts. One byproduct of Cassandra's design is that all writes are *upserts* (Section 5.3.1). If a record or a row does not exist, it is created automatically. In fact, `INSERT` and `UPDATE` are interchangeable in CQL; they perform the same operations, and only differ in syntax.

5. It is possible to store a nested collection as a blob using the `frozen` keyword. But this approach prevents accessing the entries directly via CQL.

The first upsert can only operate on static columns because the clustering column is not included. The syntax is similar to SQL, except for the JSON-like format for the map column.

Since the first upsert only provides the partition key and static row data, the `SELECT` returns one row with all `null` for the clustering and non-static data columns. Again, this behavior mirrors a `LEFT OUTER JOIN`.

The second upsert adds the first row to the partition. It is not necessary to include the static columns because they are already set. The earlier null row is replaced with a complete row.

The next upsert demonstrates the two-level nature of the partition. It is both an insert of a row, and an update of the static `Status` column to "Shipped." The subsequent `SELECT` produces two result rows, and static status information is modified in both. Again this is similar to a `LEFT OUTER JOIN`.

5.2. Cassandra's High Availability Replication

Cassandra is a peer-to-peer system. Multiple replicas (copies) of a record are distributed across the cluster. There is no particular replica that is considered the "true" copy of a record. The *replication factor* is the number of replicas of each partition in a keyspace. In Cassandra, each keyspace has a *replication strategy* that is responsible for determining where to place each replica (see Listing 3).

In deployment, the strategy takes into account network topology by setting a factor for each data center. For example, if a Cassandra cluster is deployed to 3 data centers, with 12 nodes at each data center, and the replication strategy sets the replication factor for each data center to 3, then each node holds roughly ¼ of the keyspace's partitions. For development, a simple strategy sets a cluster-wide replication factor — normally 1.

5.2.1. Eventual Consistency and Isolation Relaxation. To assure high availability, Cassandra permits any node in the cluster to handle a query, i.e. a *coordinator*. Coordinators do not rollback any upsert (write), and each replica reflects the latest upserts it has received. Therefore: a) the system is eventually consistent because replicas converge to the state of the most recent updates, and b) isolation is relaxed because concurrent coordinators can simultaneously update separate replicas of the same record at different locations.

5.2.2. The Coordinator. Cassandra assigns each CQL query to a coordinator. The coordinator is responsible for managing the query's processing.

In an upsert, each query is timestamped with the time it is received by the system. When the coordinator receives the query it sends the upsert and the timestamp as a message to all active nodes maintaining a copy of the partition. Each active node that receives the message updates its local replica, and associate the timestamp with each cell in the partition that is modified. The receiving node sends an acknowledgment to the coordinator once the change is durable.

The coordinator reports success once it receives acknowledgments from enough nodes to fulfill the conditions set by the query's *write consistency level*. For nodes where the coordinator does not receive an acknowledgment, it performs a *hinted handoff* — i.e. the coordinator retries the update once every ten minutes for up to three hours.

In handling a read, the coordinator sends messages requesting data to the active nodes maintaining replicas of the partition. The coordinator evaluates each cell using the timestamps to determine which replica has the most recent value. The coordinator informs the appropriate nodes of any stale values it received, this process is called *read repair*. After the coordinator receives enough repair acknowledgments to fulfill the query's *read consistency level*, it returns results using on the most up-to-date cell entries. After returning results, the coordinator continues with its read repair, if necessary. Finally, the coordinator performs a hinted handoff to any remaining non-responsive nodes.

The default read and write consistency levels are `ONE`. That is, if the coordinator receives a result or an acknowledgment from one node maintaining a replica it returns a success.

Therefore, Cassandra is tuned to attempt to complete a query whenever at lease one replica of the related record is available.

5.3. BASE Considerations: Quorum Consistency and Very-Likely Writes

In a peer-to-peer system, consistency does not require unanimity, only a majority. If the coordinator receives acknowledgment of an upsert to a record from a quorum, then any read of the record is visible because at least one member of a responding quorum has received the original upsert. Since, the coordinator uses the latest value for each cell, upsert is guaranteed to be visible to within quorum until it is overwritten. Thus, quorums produce consistency without the need for a central node.

Quorum based approaches are not immune to the CAP Theorem, however. To increase basic availability, Cassandra offers quorum consistency levels that are *local* quorums over a data center. Thus, consistency is possible between multiple machines at a site, where latencies are on the order of 1 ms or less [25].

5.3.1. Lightweight Transactions for Isolation. Data in Cassandra are in a volatile state because the coordinators are each acting independently. Quorum consistency does not address the issue fully because, a node may modify a local replica of a record immediately after sending the copy's contents.

Thus, Cassandra upsert queries should be idempotent, i.e. resubmission should produce the same result. Inserts are not idempotent because a second submission will fail; thus, the need for upserts in Cassandra. The assignment $x = 5$ is idempotent, as opposed to $x = x + 1$. In fact the latter expression lacks meaning since x is volatile.

In Cassandra, increments must be performed in the following manner:

1) Read x
2) Calculate $y = x + 1$
3) Read x.
4) If the value of x has not changed then set $x = y$, else return the current value of x.

To ensure another node does not change x between steps 3 and 4, Cassandra offers lightweight transactions employing a quorum-based Paxos protocol [19]. Setting the read consistency level to enforce durable *serial*

writes invokes the transactional protection; writes must be acknowledged after a quorum forms.

For similar reasons writes in Cassandra are normally upserts. Lightweight transactions are necessary to enforce true inserts and updates. Otherwise, there is no way to tell if a record exists because, it may be created or deleted elsewhere.

Using lightweight transactions makes Cassandra ACID-compliant. The CAP Theorem still applies although Paxos is a non-locking protocol. There is still a cost to availability and a risk of an unbounded wait [17].

5.3.2. Soft-State Write Timeouts. In Cassandra, a *timeout* error does not mean that a write did not occur. The coordinator sends the error to indicate that it did not receive the acknowledgments necessary to confirm that the conditions of the write consistency level were met. In most cases, a timeout write is subsequently replicated; the error is better described as an uncertain write "in-process" message [9].

A write is exceedingly unlikely to fail. First, a coordinator does not attempt a query if it is aware that the current status of the cluster cannot meet the query's consistency level; instead, it returns an *unavailable* error. In this case, the client is aware that the write did not occur. A node typically becomes aware of a fault(s) on the cluster within 15 sec [18].

Second once the coordinator does attempt the write, one or more replications may occur, but not the number required. The coordinator includes the count of the acknowledgments it received in the error message. Cassandra does not rollback a persisted write. Therefore through automated and maintenance repair processes, the write is eventually replicated.

Third, the coordinator attempts to perform hinted handoffs to any nodes that did not acknowledge its initial message — even if it received no acknowledgments. Therefore, the soft-state in Cassandra is the result of timed out write. But generally, the uncertainty lies in when, not if, the write become visible.

6. NoSQL Data Access Transparency

A data access tier hides common problems in distributed data persistence. Thus, the application tier can safely use the intermediary's interface to meet SLA requirements without a need to understand the underlying data architecture. [21]

TOGAF outlines to a set transparencies specified in the ISO Reference Model for Open Distributed Processing (RM-ODP) [21], [27] for hiding the complexities of distributed processing. This work discusses the features of MongoDB and Cassandra that are applicable to implementing six transparencies.

Eventual consistency latency benchmarks for assessing SLA requirements are based on the results for an Amazon Web Service (AWS) deployment reported in Bermbach, Zhao and Sakr [6]: a) MongoDB's results

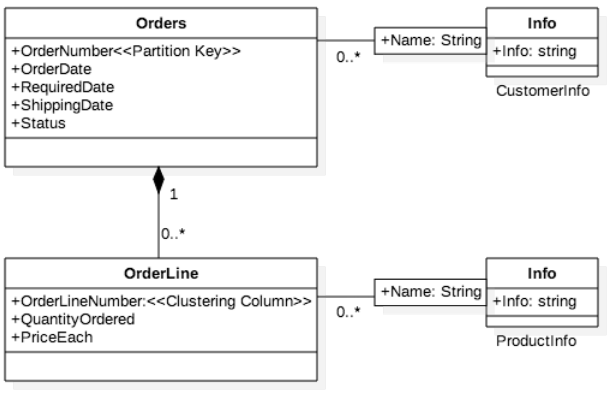

Figure 3: The Cassandra `Orders` table as an UML composition.

are 5 ms within an availability zone (roughly a data center), 15 ms within a region, and 90 ms across regions; and b) Cassandra's results are 1 ms within an availability zone, 6 ms within a region, and 110 ms across regions.

6.1. Access Transparency

Access transparency provides a language independent interface to data objects without the need to consider the logical data model. In cases where a software class can be placed in a single record, object-relational impedance mismatch is addressed.

MongoDB has native access transparency. Records are stored in a recognized language-independent protocol, JSON, capable of general-purpose data object storage. There is no need to hide the logical data model as MongoDB does not impose one.

In Cassandra tables can be matched to a UML composition in software engineering with no more than one level of collection (map, set, and list) attributes. As the representation of the customer order in Figure 3 highlights, there are many contexts in which Cassandra's one-to-many representation is sufficient.

6.2. Transaction Transparency

Both MongoDB and Cassandra adhere to the principles in Helland [16] by providing single record transactional scopes. Thus the data access tier only needs to address ACID relaxation.

MongoDB relaxes of durability to allow for a rollback of inconsistent records. Rollbacks are rare however. For rollbacks to occur, a primary server must be able to receive a write, but not to replicate it to at least one member of a replica set quorum. If a member is available within the data center, the rollback window is abut 5 ms [6]. If there is a site-wide or regional failure the window is roughly 20 seconds. However,

this scenario would likely occur under disaster recovery and business continuity SLA requirements rather than normal operations.

Cassandra's relaxation of isolation is unavoidable during normal operations. Thus, it is ideal to perform idempotent row inserts while avoiding updates and deletes. The append-only approach should be specified in documentation rather than enforced in the data access tier. To distinguish between an insert and an update requires costly lightweight transactions.

6.3. Replication Transparency

In MongoDB, replication is visible in the latency of communication with between the client and the primary member of the replica set. Thus the location of the primary member is an important consideration. During ongoing operations, it is possible to manage the location of replica set primaries. Thus the data access tier can automatically reassign primaries to follow traffic throughout the day. Each reassignment requires 10-20 seconds; thus several reassignments a day can occur without impacting SLA delivery.

Cassandra provides seamless replication transparency. The data access tier can connect to any node in the cluster. By default, writes are acknowledged after the first successful replication. The system then has multiple repair mechanisms to ensure the replication factor is met.

6.4. Failure Transparency

Failure transparency involves minimizing downtime and soft-state uncertainty. Both MongoDB and Cassandra provide mechanisms for safely hiding single machine, and site-wide, and multi-site failures.

For MongoDB, primary failover recovery is observable for roughly one minute. Recovery is possible so long as a majority of replica set members are accessible to each other, regardless of the number of data centers in the deployment. As noted, rollbacks during this process are rare, and are likely be minimal in the context of a disaster recovery or business continuity scenario.

Cassandra does not stop operations due to faults; it attempts to continue on with remaining resources. Also, if two locations lose connectivity, each will continue to act independently. Thus, aside from consistency concerns across partitions, failures are transparent. Moreover, Cassandra will continue to attempt to repair inconsistencies once connectivity is restored. Thus, in Cassandra, an acknowledged upsert is almost never lost. Soft-state issues concern when the upsert becomes visible at all locations. For example, Netflix reported it did not experience a loss of data or availability in its Cassandra deployment during an AWS outage that downed one-third of its regional nodes for three hours [23].

6.5. Migration Transparency

A data access tier with migration transparency shifts execution location within the network to optimize system responsiveness. Thus, the data store must be able to deliver highly responsive performance across the architecture. In BASE terms, migration transparency increases the tier's basic availability.

By default, MongoDB does not provide functionality for migration transparency; all operations are routed to the primary member of the replica sets. Read-only migration transparency is possible by configuring the read preference. In read-heavy workloads, changing the setting can enable fulfillment of SLA requirements. Latencies of 150 ms are possible for globally separated locations [25], while human perception is roughly 40 ms; a low latency `nearest` read can reduce access time to 5 ms or less [6].

Cassandra inherently supports migration transparency. The data access tier can connect to any portion of the cluster, and receive results from the fastest responding node(s). Netflix published benchmarks of performing over one million writes per second [8] on a roughly 300 node AWS cluster.

6.6. Relocation Transparency

Relocation transparency reflects the ability of the data access tier to hide changes to the service's execution location. Relocation transparency is inversely related to the size of the eventual consistency window.

MongoDB's CP architecture is relocation transparent by default as all access is routed to the primary. As discussed previously, the cases where relocation is visible are in read-only migration, and in failover of a primary.

While Cassandra is not relocation transparent by default, a data access tier that routes requests by a user to an assigned data center, and uses the `LOCAL_QUORUM` consistency level creates relocation transparency within the data center.

7. Summary and Conclusion

This work has presented the CAP Theorem and related BASE semantics in the context of enterprise architecture. The principles have a critical implication process modeling. To create scalable information systems architectures, process models should limit transactional scopes to entities, with messages between entities passed asynchronously.

The need to replicate data introduces the CAP theorem despite the use of single record transactions. Thus an important consequence of the CAP Theorem is that combining redundancy and consistency creates fragility in the system. Instead, systems can replicate record in a CP or AP manner and use BASE semantics to tune ACID relaxation.

NoSQL systems arose to address the limitations of join operations and referential integrity enforcement in RDBMS, along with the constraints of the CAP Theorem. Thus, common features are: a) a self-contained record structure without support for join operations, b) a single-record transactional scope, and c) a simplified CP or AP replication method with tunable BASE semantics to manage ACID relaxation.

In a technical architecture, a data access tier serves to hide the complexities of distributed persistence from one or more application tiers throughout the enterprise. Thus it is valuable to assess the NoSQL systems in the context of supporting transparency functionality.

MongoDB has proven valuable in many general-purpose applications through its role in the MEAN stack. The data access requirements in web applications frequently overlap with requirements of object repositories, particularly tiers implemented as web services.

MongoDB's transparency strengths are in: a) access transparency due to the native use of interoperable JSON records, b) transaction transparency where durability relaxation appears rarely, as a result of a site-wide failure, and c) relocation transparency due to the default CP implementation of routing all reads and writes to one primary server.

Netflix in particular has written extensively on its use of Cassandra within its enterprise. In fact Cassandra maintains Netflix's real-time live viewing data [10].

Cassandra's transparency strengths are in: a) replication transparency due to multiple repair mechanisms to ensure successful upserts, b) migration transparency is automatic as the data access tier can use any coordinator in the cluster with results returned by the fastest responding node(s), and c) failure transparency where the system continues operating after a fault with the limited capacity remaining.

Thus, MongoDB and Cassandra offer complimentary options that are conducive to developing scalable data access tiers.

References

[1] "Apache Cassandra," https://cassandra.apache.org/.

[2] "Eclipse BIRT," http://www.eclipse.org/birt.

[3] "JSON," https://www.json.org/.

[4] "Redis," http://redis.io/, accessed: 28 February,2016.

[5] M. Aslett, "NoSQL LinkedIn Skills Index – An Interesting Occasional Update," https://blogs.the451group.com/information_management/2016/12/19/nosql-linkedin-skills-index-an-interesting-occasional-update/, 2016, accessed: 11 April,2016.

[6] D. Bermbach, L. Zhao, and S. Sakr, "Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services," in *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2013, pp. 32–47.

[7] E. Brewer, "CAP twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[8] A. Cockcroft and D. Sheahan, "Benchmarking Cassandra scalability on AWS – over a million writes per second," https://medium.com/netflix-techblog/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e, 2011, accessed: 05 June,2017.

[9] J. Ellis, "When a timeout is not a failure: how Cassandra delivers high availability, part 1," http://www.datastax.com/dev/blog/how-cassandra-deals-with-replica-failure, 2012, accessed: 16 May,2017.

[10] P. Fisher-Ogden, M. Zimmer, J. Kojo, and J. Li, "Netflix's viewing data how we know where you are in House of Cards," 2015.

[11] A. Fox and E. A. Brewer, "Harvest, yield, and scalable tolerant systems," in *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*. IEEE, 1999, pp. 174–178.

[12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-based scalable network services," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 78–91, Oct. 1997. [Online]. Available: http://doi.acm.org/10.1145/269005.266662

[13] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[14] J. Gray, "The transaction concept: Virtues and limitations (invited paper)," in *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, ser. VLDB '81. VLDB Endowment, 1981, pp. 144–154. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286831.1286846

[15] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983. [Online]. Available: http://doi.acm.org/10.1145/289.291

[16] P. Helland, "Life beyond distributed transactions: an apostate's opinion." in *CIDR*, 2007, pp. 132–141.

[17] ——, "Heisenberg was on the write track." in *CIDR*, 2015.

[18] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[19] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[20] N. Leavitt, "Will NoSQL databases live up to their promise?" *Computer*, vol. 43, no. 2, 2010.

[21] P. F. Linington, Z. Milosevic, A. Tanaka, and A. Vallecillo, *Building enterprise systems with ODP: an introduction to open distributed processing*. CRC Press, 2011.

[22] MongoDB Inc., "MongoDB," https://mongodb.org/.

[23] G. Orzell and A. Tseitlin, "Lessons Netflix learned from the AWS storm," 2012.

[24] D. Pritchett, "BASE: An ACID alternative," *Queue*, vol. 6, no. 3, pp. 48–55, 2008.

[25] C. Scott, "Latency numbers every programmer should know," https://people.eecs.berkeley.edu/ rcs/research/interactive_latency.html, accessed: 31 May,2017.

[26] solid IT, "DB-Engines Ranking - popularity ranking of database management systems," http://db-engines.com/en/ranking, 2016, accessed: 28 March, 2016.

[27] The Open Group, *TOGAF Version 9.1*. The Open Group, 2011.